Programmer’s Guide to the
I3RC Community Radiative Transfer Mode

I3RC is the “Intercomparison of Three-Dimensional Radiation Code” project, funded by US National Aeronautics and Space Administration and the Atmospheric Radiation Measurement Program of the Department of Energy. See http://i3rc.gsfc.nasa.gov

The community model framework was written by Robert Pincus (Robert.Pincus@colorado.edu) with important contributions by Frank Evans (Evans@nit.colorado.edu). The documentation was written by Robert Pincus.

This version of the document describes the Bramley release, dated July, 2006.

July, 2006

Overview

The I3RC Community Monte Carlo Model is a set of code for solving radiative transfer problems in inhomogeneous cloudy atmospheres, with an emphasis on Monte Carlo methods. The basic framework is a set of modules that describe parts of a radiative transfer problem, such as the illumination source or the three-dimensional distribution of optical properties within a domain. On top of this framework we have built a rudimentary solver for a particular class of radiative transfer problems (the computation of fluxes at the top and bottom boundaries of a domain) as well as several example program and utilities.

Framework design

The package is implemented in standard Fortran 95 to strike a balance between efficiency and portability. It is coded in an “object-oriented” style that isn’t used much in the physical sciences and may take some getting used to. The basic unit of code is a Fortran 95 module. Each module in the framework is defined by one or more “objects” and a set of procedures (functions and subroutines) that operate on those objects. Each object represents a specific part of the problem - the single scattering phase function, for example. The objects are implemented as publicly-visible derived types. The components of the derived types are hidden, however, and the underlying data can only be accessed using the module’s procedures. This means that modules communicate entirely through the data structures and procedures documented in this manual. We have provided a reference implementation of the framework which we believe to be correct. Developers may replace any module in the framework with their own implementation without worry if the new version implements the data structures and procedures we have defined.

The modules in the basic framework can be used to define the radiative transfer problem at hand, but the problem is solved by a Monte Carlo “integrator” which solves the radiative transfer equation by simulating the trajectories of a large number of photons. Flexibility in one of the great strengths of the Monte Carlo method - it may be used to compute fluxes at particular levels, or directional intensities, or to keep track of photon path lengths, and so on. Each of these problems requires a different implementation of the integrator. We have provided a simple integrator module that computes fluxes and intensities at the top and bottom boundaries and estimates of column absorption. If your needs are different you will need to adapt the module we provide or write your own from scratch. Since the problem definition and the form of the solution depend on the integrator, the interface to the integrator module is flexible.

Modules in the reference implementation never read from the keyboard or write to the screen, but instead communicate through variables and procedures defined in module `ErrorMessages`. We strongly encourage other developers to do the same, since this will allow their code to be used in a wider variety of environments.

Coding notes

The framework is coded using many of the object-oriented features of Fortran 95: derived types; procedure overloading, including elemental functions; and access control through public and private interfaces and derived types. It also makes extensive use of array operations. Developers wishing to modify this code very much will need to be familiar with these ideas.

Modules define one or more derived types and procedures to act on them. Some kinds of procedures are necessary for all derived types. All modules implement the following procedures for each derived type:

- `new_` stores a definition in a new variable. A call to `new_RandomNumberSequence`, for example, initializes a random number generator from a user-provided seed and stores the random number generator state in a variable of type `randomNumberSequence`.
- `copy_` creates an identical copy of the input argument.
- `getInfo_` is used to provide information about the internal state of the variable
- `isReady_` is a boolean function indicating if the variable has been initialized
- `finalize_` release all resources used by the variable, and returns it to an un-initialized state. Some variables may use large amounts of memory or other resources, so these procedures can be very useful.

Some modules also implement `read_` and `write_` procedures which store or retrieve a copy of the variable in an external file. Implementations of a module is responsible for reading and writing its own external files.

Working with the code framework

The code framework is intended to support a wide range of radiative transfer problems, so many modules contain more functions than most users will need. The process of setting up a simple radiative transfer problem, however, isn't terribly complicated. Imagine that you want to compute the radiative transfer through a variable cloud field without considering gases, aerosols, etc. You will need to

- define the single scattering phase function for one or more drop sizes and store this in an object representing the phase function table. This is done with function `new_PhaseFunctionTable` from module `scatteringPhaseFunctions`.
- define the spatial domain of the problem using function `new_Domain` from module `opticalProperties` to create a new object of type `domain`.
- define the cloud fields by storing the three-dimensionally varying extinction and single scattering albedo in the `domain` using procedure `addOpticalComponent` from module `opticalProperties`. At the same time you'll indicate which single scattering phase function should be used at each location in the domain.
- set up the radiative transfer problem by giving the 3D cloud fields to an object of type `integrator` using procedure `new_Integrator` in module `monteCarloRadiativeTransfer` (if you're using the default implementation)
- seed the random number generator using procedure `new_RandomNumberSequence` from module `RandomNumbers`
- specify the solar illumination, if your integrator requires it, using procedure `new_PhotonStream` in module `monteCarloIllumination`
- compute the radiative transfer and record the results (procedures `computeRadiativeTransfer` and `reportResults` in the default implementation).

The following sections describe the interfaces to each module in detail, proceeding from the highest level to the lowest. To gain an overview and begin using the code it's probably better to begin with one of the example programs.

Module monteCarloRadiativeTransfer

This module computes the radiative transfer for a given three-dimensional distribution of optical properties and a specified illumination.

This module is the heart of the Monte Carlo model - it is where the desired radiative transfer quantities are computed. Monte Carlo methods solve the radiative transfer equation by simulating the trajectories of a very large number of photons. A nearly infinite number of radiative quantities can be computed from these trajectories, including photon path length distributions, hemispherically-integrated fluxes, directional intensities, and so on. We have provided a very simple integrator that computes fluxes and intnesities at the boundaries, column absorption, and the domain-mean absorption profile. Integrators for other tasks will require different kinds of input (i.e. a set of directions, if intensity is being computed) and will provide different kinds of output (i.e. the set of intensities at the desired angles). Thus the interface to this module, especially procedures `specifyParameters` and `reportResults`, should be adapted as the need arises.

Background

Monte Carlo methods solve the radiative transfer equation by sampling the trajectories of a large number of photons. In solar radiation problems, the photons are introduced with random initial position at the boundary of the domain. Their directions are chosen according to the illumination conditions.

The photons then begin their flight. A sample transmission is chosen uniformly between 0 and 1. The photon travels along its path until the cumulative extinction along the photons' path reaches the value that corresponds to this transmission. The photon then experiences a scattering event. The scattering angle is determined by choosing a random number and comparing this to the cumulative phase function. The photon's new direction is determined, a new transmission is determined, and this process repeats until the photon leaves the domain or is absorbed.

Absorption during scattering is normally modeled by multiplying the photon weight (initially 1) by the single scattering albedo of the medium doing the scattering. Radiative quantities are then computed from these weighted photons. Photons can be considered absorbed if their weight falls below some very small value.

In highly variable media a technique called "maximum cross-section" is often used to reduce the per-photon variance of flux. The cumulative extinction at each step is scaled by the maximum extinction within the domain, and the likelihood of scattering at the end of each step depends on the ratio of the local to the maximum extinction.

The atmosphere may contain more than one optically active component (e.g. aerosols, gases, and clouds often co-exist). The extinction accumulated along the photon trajectory is the extinction due to all components. Scattering may be treated either by averaging the single scattering phase functions (weighted by the extinction and single scattering albedo) or by choosing the phase function for a single component based on the relative amounts of extinction.

A technique called “Russian roulette” can help speed up calculations in absorbing media. At each scattering event, the photon weight is compared to a random number. If the weight exceeds the random number the photon trajectory is terminated, otherwise the photon weight is rest to 1. In our implementation Russian roulette is performed only when the photon weight is less than one-half.

Intensity is computed using a technique called “local estimation.” At each scattering event, the total extinction to the boundary from the scattering location is computed in each direction at which intensity is desired. The intensity at the exiting location is then incremented by the product of the transmission along that direction, the phase function of the scattering particle evaluated at the angle between the direction the photon is traveling and the intensity direction, and the photon weight.

The phase functions of large particles (i.e. those much larger than the wavelength of light being simulated) have large forward peaks due to diffraction. These peaks may be orders of magnitude larger than the phase function only a few degrees off-axis. If the forward peak happens to align with one of the directions at which intensity is being computed the contribution from an individual scattering event can be quite large, leading to large variance between calculations (and hence large error estimates). Efforts are often made to reduce this variance. We have introduced one such method here: the phase function used for local estimation can be replaced by a hybrid which replaces the original phase function at small angles by a Gaussian of user-specified width. The hybrid phase function is constructed to be continuous and properly normalized.

Using the module

Depends on modules: `ErrorMessages`, `RandomNumbers`, `numericUtilites`, `scatteringPhaseFunctions`, `inversePhaseFunctions`, `opticalProperties`, `monteCarloIllumination`.

Defines types: `integrator`.

Procedures: `new_Integrator, specifyParameters, computeRadiativeTransfer, reportResults, copy_Integrator, isReady_Integrator, finalize_Integrator.`

Procedure new_Integrator

Creates a new integrator object. This includes the copies of all the atmospheric properties within the domain; it might also include the specification of the surface.

Definition

```
function new_Integrator(atmosphere, status) result(new)
  type(domain),   intent( in) :: atmosphere
  type(ErrorMessage), intent(out) :: status
  type(integrator):: new
```

Implementation notes

In the default implementation, the surface is a Lambertian reflector with default albedo 0. This can be changed by a subsequent call to `specifyParameters`.

If the inverse phase function tables are not supplied in the definition of the atmosphere they are computed during the creation of the integrator.

Procedure specifyParameters

Specify problem- and integrator-specific parameters. The arguments to this function depend on the integrator itself. If, for example, the integrator used different values of the maximum cross-section in various layers, this routine would be used to specify the limits of those layers.

Definition

```
subroutine specifyParameters(thisIntegrator, surfaceAlbedo, useRayTracing,  &
                             minForwardTableSize, minInverseTableSize,      &
                             intensityMus, intensityPhis, computeIntensity, &
                             status)
  type(integrator),   intent(inout) :: thisIntegrator
  real,     optional, intent(in   ) :: surfaceAlbedo
  logical,  optional, intent(in   ) :: useRayTracing
  integer,  optional, intent(in   ) :: minForwardTableSize, &
                                       minInverseTableSize
  real, dimension(:), &
            optional, intent(in   ) :: intensityMus, intensityPhis
  logical,  optional, intent(in   ) :: computeIntensity, useRussianRoulette
  logical,  optional, intent(in   ) :: useHybridPhaseFunsForIntenCalcs
  real,     optional, intent(in   ) :: hybridPhaseFunWidth
  type(ErrorMessage), intent(  out) :: status
```

`thisIntegrator`	An initialized integrator object.
`surfaceAlbedo`	The albedo of the Lambertian surface. The default value is 0 (a black surface).
`useRayTracing`	Determines the algorithm used to compute the Monte Carlo radiative transfer. If `.true.` use standard Monte Carlo methods; otherwise use maximum cross-section. Default is `.false.`
`minForwardTableSize`, `minInverseTableSize`	The number of steps in the forward and inverse phase function tables. The inverse phase function table is needed to compute photon trajectories; the forward table is used only if intensities are desired. Higher values of these parameters lead to higher accuracy at the cost of more up-front computation.

`intensityMus, intensityPhis`	The directions at which the integrator should compute intensity, specified as the cosine of the zenith angle (positive values for upwelling intensity) and the azimthal angle (in degrees). The two arrays must be the same length. Intensity will be computed once these arrays are supplied .
`computeIntensity`	Flag indicating if intensity is to be computed during `computeRadiativeTransfer`. This value is ignored if `intensityMus` and `intensityPhis` are provided during the same call to `specifyParameters`.
`useRussianRoulette`	Turns Russian roulette on and off. There is little reason not to use this algorithm.
`useHybridPhaseFunsForIntenCalcs, hybridPhaseFunctionWidth`	Specifies if hybrid phase functions (as described in the introduction) are to be used for local estimation calculations of intensity. `useHybridPhaseFunsForIntenCalcs` must be set to `.true.` and `hybridPhaseFunctionWidth` to a number betweeen 0 and 30 degrees; though a more reasonable value would be 5-10 degrees.
`status`	Errors might be reported if the integrator object is not ready, if no photons are available, etc.

Procedure computeRadiativeTransfer

Processes batches of photons defined by an object of type `photonSteam` and computes the radiative quantities that result. Each call to `computeRadiativeTransfer` is considered a new calculation (i.e. previous results are discarded when this function is called).

Definition

```
subroutine computeRadiativeTransfer(thisIntegrator, &
                                    randomNumbers, incomingPhotons, status)
  type(integrator),           intent(inout) :: thisIntegrator
  type(randomNumberSequence), intent(inout) :: randomNumbers
  type(photonStream), intent(inout) :: incomingPhotons
  type(ErrorMessage), intent(  out) :: status
```

`thisIntegrator`	An initialized integrator object.
`randomNumbers`	An initialized random number stream.
`incomingPhotons`	The initial `x-y` position and direction of travel of a number of incoming photons.
`status`	Errors might be reported if the integrator object is not ready, if no photons are available, etc.

Implementation notes

Both ray-tracing and maximum cross-section solvers are available in the default implementation. They are chosen at compile time by have procedure `computeRadiativeTransfer` call either `computeRT_MaxCrossSection` or `computeRT_PhotonTracing`.

Other implementations of the integrator might wish to specify the illumination directly, rather than using `photonStream` objects. In that circumstance the illumination would probably be specified in the call to `computeRadiativeTransfer`.

Procedure reportResults

Describes the results of the last radiative transfer calculation (i.e. the last call to `computeRadiativeTransfer`). Different integrators will compute different quantities (e.g. fluxes or intensities), so this interface will vary between integrators. The interface for the default implementation is shown here.

Definition

```
subroutine reportResults(thisIntegrator, &
                         meanFluxUp, meanFluxDown, meanFluxAbsorbed, &
                         fluxUp, fluxDown, fluxAbsorbed,             &
                         absorbedProfile, volumeAbsorption,          &
                         meanIntensity, intensity,status)
  type(integrator),intent(in   ) :: thisIntegrator
  real,  optional, intent(  out) :: meanFluxUp, meanFluxDown, meanFluxAbsorbed
  real, dimension(:, :), &
         optional, intent(  out) :: fluxUp, fluxDown, fluxAbsorbed
  real, dimension(:),       optional, intent(  out) :: absorbedProfile
  real, dimension(:, :, :), optional, intent(  out) :: volumeAbsorption
  real, dimension(:),       optional, intent(  out) :: meanIntensity
  real, dimension(:, :, :), optional, intent(  out) ::     intensity
  type(ErrorMessage),                 intent(inout) :: status
```

`thisIntegrator`	An initialized integrator object.
`meanFluxUp, meanFluxDown, meanFluxAbsorbed`	Domain-mean upwelling and downwelling fluxes at the top and bottom boundaries of the domain, respectively, and domain-mean absorption within the domain.
`fluxUp, fluxDown, fluxAbsorbed`	Column-by-column values of upwelling and downwelling fluxes at the top and bottom boundaries of the domain, respectively, and domain-mean absorption within the domain.
`absorbedProfile`	Horizontally-averaged absorption as a function of height. Units are W/m^3, normalized by the specified solar flux.
`volumeAbsorption`	Absorption (flux divergence) within each grid cell.

`meanIntensity`	Domain-mean intensity at each of the angles specifed by `intenstiyMus` and `intensityPhis` in a call to `specifyParameters`.
`intensity`	Column-by-column intensity at each of the angles specifed by `intenstiyMus` and `intensityPhis` in a call to `specifyParameters`. The array is dimensioned (x, y, direction).
`status`	Errors might be reported if the integrator object is not ready, if the results arrays aren't the same size as the integrator's domain, etc.

Use

A simple Monte Carlo integration would make the following calls:

```
  mcIntegrator = new_Integrator(exampleDomain, status = status)
   randoms = new_RandomNumberSequence(seed = …)
   incomingPhotons = new_PhotonStream(solarMu, solarAzimuth, &
                                      numberOfPhotons = …,   &
                                      randomNumbers = randoms, status = status)
   call computeRadiativeTransfer(mcIntegrator, randoms, incomingPhotons, &
                                 status)
   call reportResults(mcIntegrator,                                      &
                      fluxUp(:, :), fluxDown(:, :), fluxAbsorbed(:, :), &
                      status)
```

Implementation notes

The default implementation computes the upwelling flux at the top boundary, the downwelling flux at the bottom boundary, and the flux absorbed in between; all three quantities are computed independently. These quantities are available as domain means or on a column-by-column basis.

Procedure copy_Integrator

Copies one integrator object to another. This would mostly be useful if different batches of photons were to be processed in parallel.

Definition

```
function copy_Integrator(original) result(copy)
  type(integrator), intent(in) :: original
  type(integrator) :: copy
```

Procedure isReady_Integrator

Returns `.TRUE.` if the integrator is ready to process batches of photons, i.e. if the problem has been set up correctly.

Definition

```
function isReady_Integrator(thisIntegrator)
  type(integrator), intent( in) :: thisIntegrator
  logical    :: isReady_Integrator
```

Procedure finalize_Integrator

Release the resources used by an integrator.

Definition

```
subroutine finalize_Integrator(thisIntegrator)
  type(integrator), intent(out) :: thisIntegrator
```

Implementation notes

The default version of the integrator makes copies of the three-dimensional distribution of all optical properties within the domain, so each object uses a lot of memory. That makes it especially important to finalize integrator objects when they're not needed any more.

Module opticalProperties

This module provides a representation of the three-dimensional distribution of optical properties within a domain.

Background

This module represents the spatial distribution of optical properties to be used in a radiative transfer calculation. The domain is specified as the boundaries of cells on a three-dimensional rectangular grid (i.e. the plane-parallel assumption is made).The domain holds one or more optically active components (e.g. clouds, aerosols, gases).Each component is defined within each grid cell by its single scattering properties: the values of extinction and single scattering albedo and the phase function; the latter is specified as a key into a table of phase functions. Inverse phase functions may be stored. Components may vary vertically or both vertically and horizontally, and need not fill the vertical extent of the domain.

Using the module

Depends on modules: `CharacterUtils`, `ErrorMessages`, `scatteringPhaseFunctions`, `inversePhaseFunctions`.

Defines types: `domain`.

Procedures: `new_Domain`, `addOpticalComponent`, `deleteOpticalComponent`, `replaceOpticalComponent`, `getOpticalPropertiesByComponent`, `getInfo_Domain`, `write_Domain`m `read_Domain`, `finalize_Domain`

Conventions: Physical units are not specified but must be consistent. If positions are indicated in meters, for example, extinction must be provided in inverse meters.

Procedure new_Domain

Creates a new domain composed of cells on a rectangular grid. The variables `xPosition, yPosition,` and `zPosition` denote the edges of the cells.

Definition

```
function new_Domain(xPosition, yPosition, zPosition, status)
  real,    dimension(:), intent( in) :: xPosition, yPosition, zPosition
  type(ErrorMessage),    intent(out) :: status
  type(domain)                       :: new_Domain
```

`xPosition`, `yPosition`, `zPosition`	The locations of the cell boundaries. These must be monotonically increasing.
`status`	An error occurs if any of `xPosition, yPosition, zPosition` contain values that are not unique and increasing.

Use

This call defines a domain with `nColumns` cells of equal width `deltaX` in the *x* axis, one column of width 500 along the *y* axis, and `nLayers` of depth `deltaZ` along the vertical axis.

```
stepCloud =                                                            &
  new_Domain(xPosition = deltaX * (/ 0., (real(i), i = 1, nColumns) /), &
             yPosition = (/ 0., 500.0 /),                              &
             zPosition = deltaZ * (/ 0., (real(i), i = 1, nLayers) /) , &
             status = status)
```

Procedure addOpticalComponent

Define the values of extinction, single scattering albedo, phase function, and (optionally) inverse phase function within the domain for a new optically active component. The phase function at each location is defined using an integer index into a table of phase functions; that is, if `phaseFunctionIndex(i, j, k) = l`, then the phase function within that cell is given by a call to `getElement(l, phaseFunctions, status)` (see module `scatteringPhaseFunctions`). Inverse phase functions are treated similarly: if `phaseFunctionIndex(i, j, k) = l`, then the corresponding inverse phase function is stored in the `l`th column of the array returned by `getInversePhaseFunctions`.

Optical properties are defined within cells, so the arrays that define the single scattering properties should be one smaller in each dimension than the vectors that determined the cell boundaries when `new_Domain` was called. The component need not fill the vertical extent of the domain. Arrays that are smaller than the domain requires in the vertical may be supplied; they are assumed to begin at the lowest level unless the argument `zLevelBase` is supplied.

Single scattering properties may defined with three-dimensional arrays or one-dimensional vectors; if vectors are supplied the optical component is assumed to vary only in the vertical.

Definition

```
subroutine addOpticalComponent(thisDomain, componentName,          &
                               extinction, singleScatteringAlbedo, &
                               phaseFunctionIndex, phaseFunctions, &
                               inversePhaseFunctions,              &
                                zLevelBase, status)
  type(domain),                 intent(inout) :: thisDomain
  character (len = *),          intent(in   ) :: componentName
  real,    dimension(),         intent(in   ) :: extinction, &
                                                 singleScatteringAlbedo
  integer, dimension(),         intent(in   ) :: phaseFunctionIndex
  type(phaseFunctionTable),     intent(in   ) :: phaseFunctions
  integer, optional,            intent(in   ) :: zLevelBase
  type(ErrorMessage),           intent(  out) :: status
```

`thisDomain`	The domain to which the new optical component should be added.

`componentName`	A description of the component.
`extinction,` `singleScatteringAlbedo`	The extinction and single scattering albedo within each cell in the domain. May be one or three dimensional; see the section on *Use* below.
`phaseFunctionIndex`	An integer index indicating which entry in the phase function table should be used for each cell.
`phaseFunctions`	A table of phase functions.
`zLevelBase`	Optionally, the lowest level at which the component is defined, that is, the component extends from level `zLevelBase` upward. The default value is 1.
`status`	An error occurs if arguments have the wrong sizes (i.e. three-dimensional arrays are not the same size as the underlying domain) or if they don't make sense (i.e. single scattering albedo is outside the range [0, 1]).

Use

The variables `extinction, singleScatteringAlbedo`, and `phaseFunctionIndex` define the single scattering properties of each component within the domain. The three arrays must be conformable (i.e. they must have the same extent in each dimension.) Imagine a domain defined by vectors `xPos, yPos,` and `zPos`. If the component varies only in the vertical (a gaseous absorber, say) the single scattering arrays may be provided as one-dimensional arrays of length less than or equal to `size(zPos)-1`. If the length is less than `size(zPos)-1` the component is understood not to fill the entire vertical extent of the domain, but instead extends from the level given by `zLevelBase` upwards.

Components which vary in three dimensions must fill the domain both completely in both horizontal directions (that is, `size(extinction, 1)` must be equal to `size(xPos)-1`, and similarly for the *y* direction. Components which vary in the horizontal need not fill the domain in the vertical.

Procedure deleteOpticalComponent

Releases the resources associated with a given optical component within a domain.

Definition

```
subroutine deleteOpticalComponent(thisDomain, componentNumber, status)
       type(domain),                intent(inout) :: thisDomain
       integer,                     intent(in   ) :: componentNumber
       type(ErrorMessage),          intent(  out) :: status
```

Use

The number of components held in the domain and their names can be determined by calling `getInfo_Domain`.

Procedure replaceOpticalComponent

Replaces the current values that define an optical component with new values. The component number remains the same.

Definition

```
subroutine replaceOpticalComponent3D(thisDomain, componentNumber,         &
                                     componentName,                       &
                                     extinction, singleScatteringAlbedo, &
                                     phaseFunctionIndex, phaseFunctions, &
                                     inversePhaseFunctions,              &
                                     zLevelBase, status)
      type(domain),                 intent(inout) :: thisDomain
      integer,                      intent(in   ) :: componentNumber
      character (len = *),          intent(in   ) :: componentName
      real,    dimension(:, :, :), intent(in   ) :: extinction, &
                                                    singleScatteringAlbedo
      integer, dimension(:, :, :), intent(in   ) :: phaseFunctionIndex
      type(phaseFunctionTable),    intent(in   ) :: phaseFunctions
      type(inversePhaseFunctionTable), &
                         optional, intent(in   ) :: inversePhaseFunctions
      integer, optional,           intent(in   ) :: zLevelBase
      type(ErrorMessage),          intent(  out) :: status
```

Use

This subroutine is useful for changing the optical properties of one component while the properties of the other components remain the same. It might be used, for example, to change the profiles of gaseous absorbers to do a *k*-distribution integration.

Procedure getOpticalPropertiesByComponent

Retrieves information about the domain and the optical properties it contains.

Definition

```
subroutine getOpticalPropertiesByComponent(thisDomain,            &
                                           totalExtinction,        &
                                           cumulativeExtinction,   &
                                           singleScatteringAlbedo, &
                                           phaseFunctionIndex,     &
                                           phaseFunctions,         &
                                           status)
    type(domain),                     intent( in) :: thisDomain
    real,    dimension(:, :, :),      intent(out) :: totalExtinction
    real,    dimension(:, :, :, :),   intent(out) :: cumulativeExtinction, &
```

```
                                                            singleScatteringAlbedo
  integer, dimension(:, :, :, :),  intent(out) :: phaseFunctionIndex
  type(phaseFunctionTable), &
           dimension(:), optional, intent(out) :: phaseFunctions
  type(ErrorMessage),              intent(out) :: status
```

`thisDomain`	The domain to inquire from.
`totalExtinction`	Three-dimensional field containing the sum of extinction from all components in the domain.
In the following four-dimensional arrays, the first three dimensions correspond to space (*x*-*y*-*z*) and the fourth indexes the optical component.	
`cumulativeExtinction`, `singleScatteringAlbedo`	The single scattering albedo and cumulative extinction.
`phaseFunctionIndex`	For each component, an integer index indicating which entry in the component's phase function table should be used for each cell.
`phaseFunctions`	A vector of phase function tables; element `i` is the phase function table for component `i`.
`status`	An error occurs if the arrays (`totalExtinction`, etc.) are not the same as the stored representation or if the wrong number of components are requested.

Procedure getInfo_Domain

Retrieves information about the domain, including the number of cells, their boundary positions, and the number of names of components.

Definition

```
subroutine getInfo_Domain(thisDomain, numX, numY, numZ,    &
                          xPosition, yPosition, zPosition, &
                          numberOfComponents, componentNames, status)
  type(domain),                    intent( in) :: thisDomain
  integer,               optional, intent(out) :: numX, numY, numZ
  real,    dimension(:), optional, intent(out) :: xPosition, yPosition, &
                                                  zPosition
  integer,               optional, intent(out) :: numberOfComponents
  character(len = *), &
           dimension(:), optional, intent(out) :: componentNames
  type(ErrorMessage),              intent(out) :: status
```

`thisDomain`	The domain to inquire from.
`numX, numY, numZ`	The number of cells in each dimension of the domain.

xPosition, yPosition, zPosition	The boundaries of the cells; note that the length of each of these vectors should be one larger than the number of cells (i.e. size(xPosition) = numX + 1)
numberOfComponents	The number of components stored in the domain.
componentNames	The names of the components, as provided during the call to addOpticalComponent.
status	An error occurs if the array holding the names or positions is not the same length as the number of components.

Use

This procedure might be called multiple times with different arguments. For example:

```
  !   How big is the atmospheric domain, and what are the cell boundaries?
  !
  call getInfo_Domain(atmosphere, numX, numY, numZ, &
                      numberOfComponents = numComponents,  status = status)
  allocate(xPosition(numX + 1), yPosition(numY + 1), &
           zPosition(numZ + 1))
  call getInfo_Domain(atmosphere,                       &
                      xPosition = xPosition, &
                      yPosition = yPosition, &
                      zPosition = zPosition, status = status)
```

Procedure write_Domain

Writes information about a domain, including its components, to a persistent file.

Definition

```
subroutine write_Domain(thisDomain, fileName, status)
  type(domain),       intent( in) :: thisDomain
  character(len = *), intent( in) :: fileName
  type(ErrorMessage), intent(out) :: status
```

Implementation notes

The default implementation requires Netcdf 3.5.1 or higher with Fortran 90 support. The phase function and inverse phase function tables are stored in separate files whose names are recorded in the file describing the domain.

Procedure read_Domain

Definition

```
subroutine read_Domain(fileName, thisDomain, status)
  character(len = *), intent( in) :: fileName
  type(domain),       intent(out) :: thisDomain
  type(ErrorMessage), intent(out) :: statusDefinition
```

Implementation notes

The default implementation requires Netcdf 3.5.1 or higher with Fortran 90 support. The phase function and inverse phase function tables are stored only as the names of files, which might cause problems if the domain files, phase function table files, and inverse phase function table files are not kept separate.

Procedure finalize_Domain

Releases the resources used by a domain and the optical components it contains.

Definition

```
subroutine finalize_Domain(thisDomain)
  type(domain), intent(out) :: thisDomain
```

Module monteCarloIllumination

This module provides a description of a incoming illumination for a radiative transfer problem. It is designed to describe incoming solar radiation.

Background

It is often convenient to think of Monte Carlo calculations as simulations of the trajectories of a large number of photons as they make their way through a medium. For shortwave problems in the earth's atmosphere those photons arrive from the Sun, whose position is specified in terms of the solar zenith angle μ_0 and the solar azimuthal angles ϕ_0.

For some problems it may desirable to compute averages over either or both of the polar and azimuthal angles. If the problem has no specific orientation, for example, one might want to compute the azimuthally-averaged reflection from a given atmosphere. When using Monte Carlo methods such averages may be computed by illuminating the atmosphere from all sides at once, i.e. by introducing photons from random azimuths.

This module precomputes the initial direction and position of incoming photons for a variety of commonly-used illumination conditions. Doing so simplifies the logic in the main Monte Carlo integrator at the expense of memory. The default implementation, for example, uses 16 bytes per photon. Users may choose to create small batches of photons, or to write integrators that don't use this module.

Using the module

Depends on modules: `ErrorMessages`, `RandomNumbers`.

Defines types: `photonStream`

Procedures: `new_PhotonStream`, `finalize_PhotonStream`, `getNextPhoton`, `morePhotonsExist`

Procedure new_PhotonStream

Creates a new stream of incoming photons. The illumination is specified in terms of the cosine of the solar zenith angle and the azimuth of the incoming solar beam.

Definition

```
function newPhotonStream(solarMu, solarAzimuth,                  &
                         numberOfPhotons, randomNumbers, status) result(photons)
  real,            optional, intent(in   ) :: solarMu, solarAzimuth
  integer                                  :: numberOfPhotons
  type(randomNumberSequence), intent(inout) :: randomNumbers
  type(ErrorMessage),         intent(  out) :: status
  type(photonStream)                          :: photons
```

`solarMu`	The cosine of the solar zenith angle. It may be either positive or negative. If this value is not supplied, the initial values of μ are chosen uniformly, i.e. the solar flux on the horizontal is equally weighted in μ. This is a "global" average weighting, and the day-time average is half the solar constant.
`solarAzimuth`	The azimuthal direction of the incoming photons, in degrees, supplied only if `solarMu` is specified. If `solarMu` is provided but `solarAzimuth` is not the incoming photons are introduced at a fixed zenith angle but at random azimuths.
`numberOfPhotons`	The number of incoming photons to create.
`randomNumbers`	A properly initialized variable of type `randomNumberSequence`. This is used to choose the initial positions and, if necessary, direction of each incoming photon.
`status`	This procedure reports errors or warnings if the input arguments do not make sense (i.e. the value of the phase function is less than 0 at some angle).

Use

In this example, the variable `incomingPhotons` of type `photonStream` is filled with a million photons at a solar zenith angle of 60 degrees and initial direction along the positive *x* axis.

```
   type(ErrorMessage)          :: status
  type(randomNumberSequence) :: randoms
  type(photonStream)         :: incomingPhotons
  integer, parameter         :: numPhotonsPerBatch = 1E6
  …
  randoms = new_RandomNumberSequence(seed = 100)
  incomingPhotons = new_PhotonStream(solarMu = 0.5, solarAzimuth = 0.,     &
                                     numberOfPhotons = numPhotonsPerBatch, &
                                     randomNumbers = randoms, status = status)
```

Procedure finalize_PhotonStream

Releases the resources used by an object of type photon_stream. These resources can be considerable if the number of photons is large.

Definition

```
  subroutine finalize_PhotonStream(photons)
      type(photonStream), intent(inout) :: photons
```

Use

Calling this function frees all resources used by the variable. All data is lost.

Procedure getNextPhoton

Describes the initial position and direction of the next photon available from an initialized `photonStream`. Each call to `getPhotonStream` “uses up” a photon.

Definition

```
subroutine getNextPhoton(photons, xPosition, yPosition, &
                         solarMu, solarAzimuth, status)
  type(photonStream), intent(inout) :: photons
  real,               intent(  out) :: xPosition, yPosition,&
                                       solarMu, solarAzimuth
  type(ErrorMessage),  intent(  out) :: status
```

`photons`	An initialized variable of type `photonStream`.
`xPosition, yPosition`	The `x` and `y` location of the next photon, distributed uniformly across the interval [0, 1] in both directions.
`solarMu, solarAzimuth`	The initial direction of the next photon, expressed as the cosine of the polar angle (between 0 and 1) and the azimuthal angle in degrees, measured in radians, from the positive *x* axis. The polar angle is always negative, so that the photons are moving down. These values may be fixed or may vary, depending on how `photonStream` was initialized.
`status`	This procedure reports errors if the variable `photons` has not been initialized or if the procedure is called when all the photons have been used.

Use

The following code segment might be at the beginning of the loop over each photon in the Monte Carlo integrator. It computes the initial location of the each photon in physical coordinates within the domain whose limits are `x0` to `xMax` and `y0` to `yMax`, and the initial direction, expressed as a set of three direction cosines.

```
photonLoop: do
  if(.not. morePhotonsExist(incomingPhotons)) exit photonLoop
  ! This means we've used all the photons
  call getNextPhoton(incomingPhotons, xPos, yPos, mu, phi, status)
  ! Incoming xPos, yPos are between 0 and 1.
  !   Translate these positions to the local domain
  xPos = x0 + xPos * (xMax - x0)
  yPos = y0 + yPos * (yMax - y0)
  zPos = zMax
  !
  ! Direction cosines S are defined so that
  !  S(1) = sin(theta) * cos(phi), projection in X direction
  !  S(2) = sin(theta) * sin(phi), projection in Y direction
  !  S(3) = cos(theta),            projection in Z direction
```

```
    sinTheta = sqrt(1. - mu**2)
    directionCosines(:) = (/ sinTheta * cos(phi), sinTheta * sin(phi), mu /)
    !
    ! Loop over the order of scattering
    …
  end do photonLoop
```

Procedure morePhotonsExist

Indicates whether a variable of type `photonStream` has any more photons left.

Definition

```
function morePhotonsExist(photons)
  type(photonStream), intent(inout) :: photons
  logical                           :: morePhotonsExist
```

Use

See the example for procedure `getNextPhoton`.

Module RandomNumbers

This module provides a uniform interface to random number generators.

Background

Monte Carlo methods are based on the ability to generate long sequences of pseudo-random numbers. The generators must be "seeded" an some initial value (which is not the same as the first random number in the sequence). We encapsulate the state of the random number generator so that different streams of random numbers can be generated in parallel.

Fortran 90 include random number generation, but these implementations aren't parallel and depend on the platform. We don't recommend using them.

Implementation notes

In the current implementation, the random numbers are generated with a Mersenne Twister (`http://www.math.keio.ac.jp/matumoto/emt.html`).

Using the module

Depends on modules: none.

Defines types: `randomNumberSequence`

Procedures: `new_RandomNumberSequence, finalize_RandomNumberSequence, getRandomReal`

Procedures

Procedure new_RandomNumberSequence

Creates a new random number sequence from a scalar or vector integer seed.

Definition

```
function new_RandomNumberSequence(seed) result(randoms)
  ! seed may be either
  integer,              intent( in) :: seed
  ! or
  integer, dimension(:), intent( in) :: seed
  type(RandomNumberSequence)         :: randoms
```

Use

Random number streams must be initialized ("seeded") before they can be used. The integer seed may be a scalar or a vector of arbitrary length.

```
  randoms = new_RandomNumberSequence(seed = (/ 10, 100, 1000 /) )
```

Procedure finalize_RandomNumberSequence

Releases resources used by a variable of type `RandomNumberSequence` and returns the variable to an un-initialized state.

Definition

```
  subroutine finalize_RandomNumberSequence(randoms)
    type(RandomNumberSequence), intent(out) :: randoms
```

Procedure getRandomReal

Gets the next real number on the interval [0, 1] from a previously-initialized sequence of random numbers.

Definition

```
function getRandomReal(randoms)
  type(RandomNumberSequence), intent(inout) :: randoms
  real                                      :: getRandomReal
```

Use

Random numbers are generated one at a time; if you want more than one you must write a loop.

```
  do i = 1, numberOfRandoms
    randomNumbers(i) = getRandomReal(randoms)
  end do
```

Module inversePhaseFunctions

This module computes the inverse scattering phase functions from a set of phase functions stored in a variable of type `phaseFunctionTable` from module `scatteringPhaseFunctions`. The inverse phase functions are returned as a 2D array. The first dimension of length *N* entries; entry *i* contains the scattering angle at which the integral of the phase function has the normalized value *i*/*N*. The second dimension corresponds to the variation according to some key parameter (i.e. drop size); it is. The inverse phase function table simplifies the calculation of the scattering angle at each scattering event during a Monte Carlo calculation.

Background

Monte Carlo methods solve the radiative transfer equation by simulating the trajectories of a large number of photons. At each scattering event the photon's new direction must be determined from the single scattering phase function $P(\theta)$. This can be accomplished through the cumulative integral $C(\theta)$ of the phase function

$$C(\theta) = \int_0^\theta P(\theta)d\theta \tag{1}$$

$C(\theta)$ varies between 0 and 1 if the scattering phase function is normalized. The new scattering direction can be found by choosing a random number *r* from a uniform distribution between 0 and 1, then finding the value of θ at which $C(\theta) = r$. In practice, it is often easiest to pre-compute $\theta(r)$ for a large number of equally-spaced values of *r* and store the results in a table.

This module provides a way to compute the cumulative phase functions in a consistent way.

Using the module

Depends on modules: `scatteringPhaseFunctions`, `numericUtilites`, `ErrorMessages`.

Procedures: `computeInversePhaseFuncTable`

Conventions: Angles are expressed in radians.

Procedure computeInversePhaseFuncTable

Computues a table of inverse phase functions from

Definition

```
subroutine computeInversePhaseFuncTable(forwardTable, inverseTable, status) &

  type(phaseFunctionTable),      intent(in   ) :: forwardTable
  real, dimension(:, :),         intent(  out) :: inverseTable
  type(ErrorMessage),             intent(inout) :: status
```

`forwardTable`	A variable of type `phaseFunctionTable` as defined in module `scatteringPhaseFunctions`.
`inverseTable`	The inverse phase function table. The first dimension is the number of equal probability intervals, so that entry *i* of the total number *N* contains the scattering angle corresponding to $C = i/N$. The second is the number of entries in the phase function table.
`status`	This procedure reports errors or warnings if the input arguments do not make sense (i.e. arrays are the wrong sizes, variables are not initialized).

Module scatteringPhaseFunctions

This module represents the results of single scattering calculations: the single scattering phase function and, optionally, the extinction and single scattering albedo. There are facilities to represent the results of individual calculations (i.e. a single drop size distribution) or a set of such calculations.

Background

This module represents single scattering calculations, especially the single scattering phase function $P(\Theta)$ or a collection of such phase functions. The phase function describes how light incident on a volume is redistributed with respect to the scattering angle Θ or the cosine of this angle $\mu = \cos\Theta$. The phase function is normalized such that:

$$\int_0^1 P(\mu) d\mu = 1 \tag{1}$$

Phase functions are frequently expressed in a Legendre polynomial expansion

$$P(\mu) = \sum_{l=0}^{L} (2l+1)\chi_l P_l(\mu) \tag{2}$$

where $P_l(\mu)$ are the Legendre polynomials (defined on the interval from 0 to 1) and χ_l are the coefficients in the expansion. Scattering phase functions are determined by single scattering calculations, which might also predict the extinction and single scattering albedo. These latter quantities may optionally be stored in variables of type `phaseFunction` and `phaseFunctionTable`.

This module contains parallel representations for single phase functions and for tabulated sets of phase functions. The tables might be used, for example, to represent how the phase function varies with particle size.

Using the module

Depends on modules: `numericUtilites`, `errorMessages`.

Defines types: `phaseFunction`, `phaseFunctionTable`.

Procedures: `new_PhaseFunction, copy_PhaseFunction, getInfo_PhaseFunction, finalize_phaseFunction, isReady_phaseFunction, getPhaseFunctionValues, getPhaseFunctionCoefficients, getExtinction, getSingleScatteringAlbedo, new_PhaseFunctionTable, copy_PhaseFunctionTable, getInfo_PhaseFunctionTable, finalize_phaseFunctionTable, isReady_phaseFunctionTable, getElement, write_PhaseFunctionTable, read_PhaseFunctionTable`

Conventions: Scattering angles are in radians, ascending from 0 (forward scattering) to π (back scattering). Coefficients in the Legendre expansion don't include the 2 l + 1 factor.

Procedures for individual phase functions

Procedure new_PhaseFunction

Creates a new phase function definition, specified either as a set of coefficients in a Legendre polynomial expansion, or as a set of angles and the value of the phase function at each angle.

Definition

```
function new_PhaseFunction(phaseFunctionDefinition,             &
                           extinction, singleScatteringAlbedo, &
                            description, status) result(newPhaseFunction)
  ! phaseFunctionDefinition may be either
  real, dimension(:),           intent( in) :: value, scatteringAngle
  ! or
  real, dimension(:),           intent( in) :: legendreCoefficients
  ! Other arguments are
  real,               optional, intent( in) :: extinction, &
                                               singleScatteringAlbedo
  character(len = *), optional, intent( in) :: description
  type(ErrorMessage),           intent(out) :: status
  type(phaseFunction)                       :: newPhaseFunction
```

phaseFunctionDefinition may be specified in one of two mutually exclusive ways.	
`value`, `scatteringAngle`	Two arrays of equal length. `scatteringAngle` is specified in radians, and must increase from 0 (forward scattering) to π (back scattering). `value` is the value of the phase function at each angle. The phase function is normalized according to (1) using trapezoidal quadrature.
`legendreCoefficients`	The coefficients in a Legendre polynomial expansion of the phase function.
`extinction`, `singleScatteringAlbedo`	The extinction and single scattering albedo, in user-defined units.
description	Annotation of the user's choice.
`status`	This procedure reports errors or warnings if the input arguments do not make sense (i.e. the value of the phase function is less than 0 at some angle).

Use

In this example, the variable `phase` of type `phaseFunction` is used to store a Henyey-Greenstein phase function.

```
  phase = new_PhaseFunction(g**(/ (i, i = 1, nLegendreCoefficients )/), &
                            status = status)
```

Procedure copy_PhaseFunction

Creates a copy of an existing phase function.

Definition

```
function copy_PhaseFunction(original) result(thisCopy)
     type(phaseFunction), intent( in) :: original
     type(phaseFunction)              :: thisCopy
```

Use

If `p1` and `p2` are both variables of type `phaseFunction`, and `p1` has been previously defined, its contents can be copied to `p2`:

```
  p2 = copy_PhaseFunction(p1)
```

Procedure getInfo_PhaseFunction

Provides information about the native representation of a particular phaseFunction variable.

Definition

```
subroutine getInfo_PhaseFunction(phaseFunctionVar, nCoefficients, nAngles, &
                                 nativeAngles, status)
  type(phaseFunction),           intent( in) :: phaseFunctionVar
  integer,             optional, intent(out) :: nCoefficients, nAngles
  real, dimension(:),  optional, intent(out) :: nativeAngles
  type(ErrorMessage),  optional, intent(out) :: status
```

`nCoefficients,` `nAngles`	The number of Legendre coefficients or the number of angles at which the phase function was originally defined. If the phase function was originally defined as a set of Legendre coefficients then `nAngles`=0; if the phase function was defined as a set of angle-value pairs then `nCoefficients = 0.`
`nativeAngles`	The angles at which the phase function was originally defined (i.e. the value of the argument `scatteringAngle` supplied when `new_PhaseFunction` was called).
`status`	An error occurs if the input phase function has not been defined. A warning is set if the length `nativeAngles` does not match original definition.

Use

This procedure can be used in conjunction with `getPhaseFunctionValues` or `getPhaseFunctionCoefficients` to recover the phase function as it was originally defined. In this example `p1` is a variable of type `phaseFunction` and `angles`, `values`, and `coeffs` are real, allocatable arrays:

```
  call getInfo_PhaseFunction(p1, nCoefficients, nAngles, status = status)
  if (.not. stateIsFailure(status)) then
    if(nAngles > 0) then
      allocate(angles(nAngles), values(nAngles))
      call getInfo_PhaseFunction(p1, nativeAngles = angles, status = status)
      call getPhaseFunctionValues(p1, angles, values, status)
    else
      allocate(coeffs(nCoefficients))
      call getPhaseFunctionCoefficients(p1, coeffs, status)
    end if
  end if
```

Procedure finalize_phaseFunction

Releases resources used be a variable of type `phaseFunction`. This procedure should be called before the variable is reused.

Definition

```
subroutine finalize_PhaseFunction(phaseFunctionVar)
    type(phaseFunction), intent(inout) :: phaseFunctionVar
```

Use

Calling this function frees all resources used by the variable. All data is lost.

```
  call finalize_phaseFunction(p1)
```

Procedure isReady_phaseFunction

Tests a phase function variable to see if it holds a valid phase function definition

Definition

```
elemental function isReady_phaseFunction(phaseFunctionVar)
  type(phaseFunction), intent( in) :: phaseFunctionVar
  logical                          :: isReady_phaseFunction
```

The function returns `.TRUE.` the phase function variable is ready for use, and `.FALSE.` otherwise.

Procedure getPhaseFunctionValues

Retrieves the value of the phase function at a set of arbitrary angles. If the phase function was originally stored as angle-value pairs the value at arbitrary angles is computed by interpolating linearly in the cosine of the scattering angle.

Definition

Phase function values may be determined from either a single phase function or from a phase function table.

```
subroutine getPhaseFunctionValues(phaseFunctionVar, scatteringAngle, value, &
                                  status)
    type(phaseFunction), intent(in   ) :: phaseFunctionVar
    real, dimension(:),  intent(in   ) :: scatteringAngle
    real, dimension(:),  intent(  out) :: value
    type(ErrorMessage),  intent(inout) :: status
```

or

```
subroutine getPhaseFunctionValues(table, scatteringAngle, values, status)
    type(phaseFunctionTable), intent(in   ) :: table
    real, dimension(:),       intent(in   ) :: scatteringAngle
    real, dimension(:, :),    intent(  out) :: values
    type(ErrorMessage),       intent(inout) :: status
```

Either a single phase function or a table of phase functions may be supplied	
`phaseFunctionVar`	The phase function. It may have been originally defined as either a set of angles and values or as a set of Legendre coefficients.
`table`	A phase function table
`scatteringAngle`	The angles, specified in radians, at which the value of the phase function(s) is/are desired. These must be increasing and unique. If the phase function was originally specified as angle-value pairs, the value is interpolated to the desired angles linearly in the cosine of the scattering angle.
The dimensions of the output array depend on the kind of the input argument:	
`value`	The value of the normalized phase function at the angles specified by `scatteringAngle`.
`values`	The values of the normalized phase function at the angles specified by `scatteringAngle`. The first dimension of the array is the same as the length of `scatteringAngle`, and the second dimension as large as the number of entries in the phase function table.
`status`	The procedure fails if any of the input arguments are inappropriate.

Use

In this example we create a Henyey-Greenstein phase function using 64 Legendre coefficients, then retrieve the value of this phase function at one degree intervals from 0 to 180 degrees.

```
 integer, parameter :: nAngles = 181
real,    parameter :: g = 0.85 ! Asymmetery parameter
real, dimension(nAngles) :: scatteringAngle, phaseFunctionValue
…
! Henyey-Greenstein phase function
phase = new_PhaseFunction(g**(/ (i, i = 1, 64 )/), status = status)
! Scattering angles every one degree from 0 to 180, expressed in radians
scatteringAngle(:) = pi / 180. * (/ (i, i = 1, nAngles) /) )
call getPhaseFunctionValues(phase, scatteringAngle, value, status)
```

Procedure getPhaseFunctionCoefficients

Retrieves the coefficients χ_i in the Legendre expansion of the phase function (2), starting with χ_1 (since $\chi_0 \equiv 1$). Note that, if the phase function was supplied as Legendre coefficients, the number of coefficients stored in the variable is available through procedure `getInfo_PhaseFunction`.

Definition

```
subroutine getPhaseFunctionCoefficients(phaseFunctionVar, &
                                         legendreCoefficients, status)
    type(phaseFunction), intent( in) :: phaseFunctionVar
    real, dimension(:),  intent(out) :: legendreCoefficients
    type(ErrorMessage),  intent(out) :: status
```

`phaseFunctionVar`	The phase function. It may have been defined as either a set of angles and values or as a set of Legendre coefficients.
`legendreCoefficients`	The coefficients in the Legendre series expansion of the phase function. If the phase function was originally defined as a series of `N` coefficients, then `legendreCoefficients(N+1:) = 0.`
`status`	The procedure fails if any of the input arguments are inappropri-ate.

Use

The following example shows how to read the coefficients of the phase function expansion, assuming that the variable `phase` has been initialized at some point in the code.

```
 integer, parameter :: nCoefficients = 128
real, dimension(nCoefficients) :: coefficients
…
call getPhaseFunctionCoefficients(phase, coefficients, status)
```

Implementation Notes

Important: In the current implementation, the Legendre coefficients are not computed if the phase function was originally stored as a set of values at specified scattering angles.

Procedure getExtinction

Retrieves the extinction, in user-defined units, supplied when the phase function was initialized.

Definition

```
elemental function getExtinction(phaseFunctionVar)
```

This is a real-valued function. If no value of extinction was supplied when the phase function was created this function returns 0.

Procedure getSingleScatteringAlbedo

Retrieves the single scattering albedo supplied when the phase function was initialized.

Definition

```
elemental function getSingleScatteringAlbedo(phaseFunctionVar)
```

This is a real-valued function. If no value of single scattering albedo was supplied when the phase function was created this function returns 0.

Procedures for collections ("tables") of phase functions

A variable of type `phaseFunctionTable` is a collection of individual phase functions with a key that describes the parameter that varies among the phase functions. If you have computed a set of phase functions for cloud drops as a function of drop size, for example, they can be collected in a phase function table with `key` indicating the effective radius. Phase function tables are built on individual phase functions, so most of what applies to variables of type `phaseFunction` also applies to variables of type `phaseFunctionTable`. Phase function tables normally describe the dependence of the single scattering properties of a single component of the atmosphere (e.g. aerosols) depend on some parameter (e.g. particle size).

Procedure new_PhaseFunctionTable

Creates a new phase function table. The phase functions may be specified as an arbitrary vector of type `phaseFunction`, or may be a set of phase function values at a uniform set of scattering angles.

Definition

```
function new_PhaseFunctionTable(phaseFunctionDefinitions,              &
                  phaseFunctionDescriptions, tableDescription, status)&
                  result(newPhaseFunction)
   ! phaseFunctionDefinitions may be either
```

```
  real, dimension(:),               intent( in) :: scatteringAngle
  real, dimension(:, :),            intent( in) :: values
  real, dimension(:), optional,     intent( in) :: extinction, &
                                                   singleScatteringAlbedo
 ! or
  type(phaseFunction), dimension(:),intent( in) :: phaseFunctions
 ! Other arguements are
  real, dimension(:),               intent( in) :: key
  character(len = *), optional,     intent( in) :: tableDescription
  type(ErrorMessage),               intent(out) :: status
  type(phaseFunctionTable)                      :: table
```

`phaseFunctionDefinitions` may be specified in one of two mutually exclusive ways. To specify a set of phase function values at a uniform set of scattering angles:	
`scatteringAngle`	The set of scattering angles for a collection of phase functions
`values`	A 2D array containing the phase functions for each entry in the variable `key`. The first dimension is the scattering angle (i.e. `size(values, 1)` must equal `size(scatteringAngle)`); the second corresponds to the key (i.e. `size(values, 2)` must equal `size(key)`).
`extinction`, `singleScatteringAlbedo`	Optionally, the extinction and single scattering albedo, in user-defined units. These must be the same length as `key`. The default value is 0.
More generally:	
`phaseFunctions`	A vector of variables of type `phaseFunction`, defined arbitrarily. This must be the same length as `key`.
In both cases:	
`key`	The value of the parameter that varies among the phase functions. If the various entries represent the way phase function varies with drop size, for example, then `key` holds the drop size for each entry. The values should be unique and increasing.
`phaseFunctionDescriptions`	Optionally, the descriptions of the individual phase functions.
`tableDescription`	Optionally, a description of the entire table (i.e. “Hexagonal crystals vs. size, computed using ray-tracing”).
`status`	Reports errors or warnings if one or more input arguments do not make sense, have the wrong size, etc.

Use

Imagine that we had written a procedure called `Mie` to compute the phase function of a distribution of water drops as a function of their effective radius, and have defined a set of scattering angles at which we want the phase function values. We can store the phase functions for drop sizes from 1 to 20 microns as follows:

```
integer, parameter :: nSizes = 20, nAngles = 1000
 real, dimension(nAngles), parameter :: scatteringAngles = (/ … /)
 real, dimension(nSizes) :: key
 real, dimension(nSizes, nAngles) :: values

 do size = 1, nSizes
   values(i, :) = Mie(real(size), scatteringAngles)
 end do
 table = new_PhaseFunctionTable(scatteringAngles, values,                 &
                                key = real( (/ (size, size = 1, nSizes) /) ), &
                                  status)
```

Procedure copy_PhaseFunctionTable

Creates a copy of an existing phase function table. Analogous to `copy_PhaseFunction`.

Definition

```
function copy_PhaseFunctionTable(original) result(thisCopy)
     type(copy_PhaseFunctionTable), intent( in) :: original
     type(copy_PhaseFunctionTable)              :: thisCopy
```

Use

If `t1` and `t2` are both variables of type `phaseFunctionTable`, and `t1` has been previously defined, its contents can be copied to `t2`:

```
t2 = copy_PhaseFunctionTable(t1)
```

Procedure getInfo_PhaseFunctionTable

Provides information about a variable of type `phaseFunctionTable`.

Definition

```
subroutine getInfo_PhaseFunctionTable(table, nEntries, key, &
                              tableDescription, status)
  type(phaseFunctionTable),           intent( in) :: table
  integer,                  optional, intent(out) :: nEntries
  real,    dimension(:),    optional, intent(out) :: key,extinction, &
                                                     singleScatteringAlbedo
  character(len = *), &
           dimension(:),    optional, intent(out) :: phaseFunctionDescriptions
  character(len = *),       optional, intent(out) :: tableDescription
  type(ErrorMessage),                  intent(out) :: status
```

`nEntries`	Optionally, the number of phase functions in the table.
`key`	Optionally, the value of `key` supplied when the table was created.
`extinction, singleScatteringAlbedo`	Optionally, the extinction and single scattering albedo as a function of the key value. Extinction is in user-specifed units of inverse length.
`phaseFunctionDescriptions`	Optionally, text descriptions of the individual phase functions.
`tableDescription`	Optionally, the description of the entire table.
`status`	An error occurs if the table has not been defined.Warnings are set if `key` or `phaseFunctionDescriptions` are not the same length as the number of entries in the table.

Procedure finalize_phaseFunctionTable

Releases resources used be a variable of type `phaseFunctionTable`. Analogous to `finalize_phaseFunction`.

Procedure isReady_phaseFunctionTable

Tests a phase function variable to see if it holds a valid phase function table. Analogous to `isReady_phaseFunction`.

Procedure getElement

Extracts a single phase function from a `phaseFunctionTable`. The resulting `phaseFunction` variable is in the same form (i.e. Legendre coefficients or angle-value pairs) in which the table was originally defined.

Definition

```
function getElement(n, table, status)
  integer,                  intent( in) :: n
  type(phaseFunctionTable), intent( in) :: table
  type(ErrorMessage),       intent(out) :: status
  type(phaseFunction)                   :: getElement
```

Use

To extract the fifth element of a phase function table

```
  phase = getElement(5, table, status)
```

Errors occur if the table is not initialized or if a non-existent entry is requested.

Procedure write_PhaseFunctionTable

Writes a set of phase functions and the information that describes them to a disk file for later use.

Definition

```
subroutine write_PhaseFunctionTable(table, fileName, status)
  type(phaseFunctionTable), intent( in) :: table
  character(len = *),       intent( in) :: fileName
  type(ErrorMessage),       intent(out) :: status
```

Use

Frank Evans at the University of Colorado distributes programs that compute the single scattering properties for cloud water drop distributions with variable effective radii. The phase function for each phase function is written to a text file as the shortest possible series of Legendre coefficients, though Frank includes the $2l + 1$ term in his coefficients he stores. The main loop of a utility program to convert his files to this format and write them out is as follows:

```
  do i = 1, nEntries
    read(fileUnit, *) key(i), extinction, ssa, nTerms
    allocate(legendreCoefficients(nTerms - 1))
    read(fileUnit, *) legendre0, legendreCoefficients(:)
    ppVector(i) = new_PhaseFunction(legendreCoefficients / &
                                (/ (2 * l + 1, l = 1, nTerms) /), &
                                extinction, ssa, status = status)
    call printStatus(status)
    deallocate(legendreCoefficients)
  end do
  table = new_PhaseFunctionTable(ppVector, key, tableDescription, status)
  call printStatus(status)
  call write_PhaseFunctionTable(table, trim(outputFileName), status)
```

Implementation notes

In the current implementation, tables may only be written if the phase function table was specified using a single set of angles, or as a set of Legendre coefficients for each phase function.

The current implementation requires Netcdf 3.5.1 or higher with Fortran 90 support.

Procedure add_PhaseFunctionTable

Adds a phase function table to an open file. A character string may be used to uniquely identify tables within a file.

Definition

```
subroutine add_PhaseFunctionTable(table, fileId, prefix, status)
  type(phaseFunctionTable),   intent( in) :: table
  integer,                    intent( in) :: fileId
  character(len=*), optional, intent( in) :: prefix
  type(ErrorMessage),         intent(out) :: status
```

Implementation notes

This function relies on implementation detils in that the file referenced by `fileId` must be of the same type as is written internally by the `scatteringPhaseFunctions` module. In the default implementation, for example, `fileId` must refer to an open netCDF file in data mode. The function is normally used to bundle phase function table with the rest of the components when storing a variable of type `domain` from module `opticalProperties`.

Procedure read_PhaseFunctionTable

Reads a set of phase functions and their descriptive information from a disk file written by `write_PhaseFunctionTable`. If the table has been previously defined, it should be released with `finalize_phaseFunctionTable` before being passed to `read_PhaseFunctionTable`**.**

Definition

```
subroutine read_PhaseFunctionTable(fileName, fileId, table, prefix, status)
  character(len = *), optional, intent( in) :: fileName
  integer,            optional, intent( in) :: fileId
  type(phaseFunctionTable),     intent(out) :: table
  character(len = *), optional, intent( in) :: prefix
  type(ErrorMessage),           intent(out) :: status
```

The phase function table is read from a file specified in one of two ways:	
`fileName`	The name of the file from which the table is to be read
or	
`fileId`	The numeric ID of the file from which the table is to be read. The file associated with this ID must be open and readable using the same methods as the calling procedure.
`table`	The phase function table.
`prefix`	Optionally, a character string used to uniquely identify each phase function table to be read from the file.
`status`	An error occurs if the table can not be read.

Implementation notes

Supply `fileName` if the table is to be read from a stand-alone file. The capability to read from an open file using `fileID` is used when the phase function tables are stored along with other information (as in objects of type `domain` in module `opticalProperties`, for example).

In the current implementation, tables may only be read if the phase function table was specified using a single set of angles, or as a set of Legendre coefficients for each phase function.

The current implementation requires Netcdf 3.5.1 or higher with Fortran 90 support.

Module ErrorMessages

This module provides a means for procedures to signal whether they have succeed, encountered difficulties, or failed during execution. The chain of events can be saved and reported.

Background

Unlike some other programming languages (e.g. Java), Fortran does not provide any graceful way to handle error conditions. Typically, error are handled by printing messages to the screen and perhaps halting execution, or by setting some integer variable to a predefined value. But printing to the screen may not be desirable - if code is running within a graphical user interface, for example, (as part of a Macintosh application, say) then "the screen" may not exist. Simply indicating that some calculation just hasn't succeeded, on the other hand, can make debugging very difficult.

This module provides a uniform way for procedures to accumulate a history of which steps have succeeded, encountered problems, or failed. A variable of type `ErrorMessage` encapsulates the "state" of the calculation, which may be one of `Success`, `Warning`, or `Failure`. Text messages can be added each time the state is set. Setting the state appends information to the variable, and the current state can be determined at any time. As the variable is passed from procedure to procedure, therefore, a history is constructed. This allows for more informative reporting - if function `A` depends on the results of functions `B` and `C`, and function `B` fails, function `A` can note why it did not complete when returning a failure state to its calling routine.

The state of an variable of type `ErrorMessage` whose state has not been explicitly set is not defined.

Using the module

Depends on modules: none.

Defines types: `ErrorMessage`

Procedures: `initializeState`, `setStateToSuccess`, `setStateToWarning`, `setStateToFailure`, `setStateToCompleteSuccess`, `stateIsSuccess`, `stateIsWarning`, `stateIsFailure`, `firstMessage`, `nextMessage`, `getCurrentMessage`, `moreMessagesExist`, `getErrorMessageLimits`

Procedures for setting and testing the current state

Procedure initializeState

Erases any information held in the variable and sets the state to `Undefined`.

Definition

```
subroutine initializeState(messageVariable)
  type (ErrorMessage),          intent (out) :: messageVariable
```

Procedure setStateToSuccess, setStateToWarning, setStateToFailure, setStateToCompleteSuccess

Sets the state to one of `Success`, `Warning`, or `Failure`. Optional text messages may be included. A call to `setStateToCompleteSuccess` removes any existing history.

Definition

```
subroutine setStateTo[Success|Warning|Failure](messageVariable, messageText)
  type (ErrorMessage),           intent (out) :: messageVariable
  character (len = *), optional, intent ( in) :: messageText
```

Use

Here is an example in which function a fails if function b or c fails.

```
  function a(…, status)
    …
    type(ErrorMessage), intent(inout) :: status
    …
    call b(…, status)
    if (stateIsSuccess(status)) then
      …
      call c(…, status)
      if (stateIsSuccess(status)) then
        …
      else
        call setStateToFailure(status, “Function a: function c failed.”)
      end if
    else
      call setStateToFailure(status, “Function a: function b failed.”)
    end if
    if(stateIsSuccess(status)) call setStateToSuccess(status)
  end function a
```

Procedure stateIsSuccess, stateIsWarning, stateIsFailure

Determine whether current state: `stateIsSuccess` returns `.true.` if the current state is `Success`, `stateIsWarning` returns `.true.` if the current state is `Warning`, etc. If the state has never been set it is undefined and all three functions will return `.false.`

Definition

```
function stateIs[Success|Warning|Failure](messageVariable)
  type (ErrorMessage), intent ( in) :: messageVariable
  logical                           :: stateIsSuccess
```

Procedures for setting and testing the current state

A variable of type `ErrorMessage` may be thought of as a list or stack of states and associated error messages. The following four procedures are used together to work through the history of states.

This follows the “Iterator” pattern from the book “Design patterns: Elements of reusable object-oriented software” (E Gamma, R. Helm, R. Johnson, and J. Vlissides, 1995, ISBN 0-201-63361-2). It is more robust than determining the number of states in the history and marching trough them one by one, since the number of states in the history may be changed at any time without ill effect.

Procedure firstMessage

Makes the first state set for this variable (typically the state set by the lowest level of the call tree) the current state.

Definition

```
subroutine firstMessage(messageVariable)
  type (ErrorMessage), intent (inout) :: messageVariable
```

Use

The following subroutine starts at the beginning of the history and prints out the state at each step, as well as any text messages that were supplied.

```
  subroutine printStatus(status)
    use ErrorMessages
    type(ErrorMessage), intent(inout) :: status

    call firstMessage(status)
    historyLoop: do
      if (.not. moreMessagesExist(status)) exit historyLoop
      ! Only one of these lines will be executed
      if(stateIsSuccess(status)) print *, “Success:”
      if(stateIsWarning(status)) print *, “Warning:”
      if(stateIsFailure(status)) print *, “Failure:”
      if(len_trim(getCurrentMessage(status)) > 0) &
        print *, trim(getCurrentMessage(status))
      call nextMessage(status)
    end do historyLoop
   end subroutine printStatus
```

Procedure nextMessage

Makes the next state set for this variable (i.e. the next event in the history) the current state.

Definition

```
subroutine nextMessage(messageVariable)
  type (ErrorMessage), intent (out) :: messageVariable
```

Use

See the example for procedure `firstMessage`.

Procedure getCurrentMessage

Returns the text message associated with the current state. If no message was supplied the function returns the null string (“”).

Definition

```
function getCurrentMessage(messageVariable)
  type (ErrorMessage),    intent (in)  :: messageVariable
  character (len = max_message_length) :: getCurrentMessage
```

Use

See the example for procedure `firstMessage`.

Procedure moreMessagesExist

Returns `.true.` if there are elements in the history beyond the current element and `.false.` otherwise.

Definition

```
function moreMessagesExist(messageVariable)
  type (ErrorMessage), intent (in) :: messageVariable
  logical                          :: moreMessagesExist
```

Use

See the example for procedure `firstMessage`.

Procedure getErrorMessageLimits

Definition

```
subroutine getErrorMessageLimits(messageVariable, maxNumberOfMessages, &
                                  maxMessageLength)
  type (ErrorMessage), intent ( in) :: messageVariable
  integer, optional,   intent (out) :: maxNumberOfMessages, maxMessageLength
```

Reports the maximum number of messages that may be stored and the maximum number of characters in each message. These parameters are set by the module developers, who should ensure that both are big enough not to cause problems.

Implementation note

In the default implementation, `maxNumberOfMessages = 100` and `maxMessageLength = 256`.

Module numericUtilites

This module contains numeric utilities: subroutines to compute the weights and abscissae for Lobatto and Gauss-Legendre quadrature; the value of the Legendre polynomials at specified values, and a routine to find the two entries in a table that bracket a test value.

Background

Lobatto and Gauss-Legendre quadrature are methods for integrating functions on known interval ([-1, 1] and (-1, 1), respectively). For N point quadrature the methods supply a set of points in the interval called the abscissae x_N and weights $w_N(x_N)$. An arbitrary function $f(x)$ can then be approximated as

$$\int_{-1}^{1} f(x)d(x) \cong \sum_{i=1}^{N} f(x_i)w_i(x_i) \tag{1}$$

These quadrature methods are accurate to higher order than, say, the trapezoidal rule.

The Legendre polynomials P_l are a set of orthogonal, normalized polynomials on the interval [-1, 1]. A series of Legendre polynomials are often used to express the dependence of the single scattering phase function on the cosine of the scattering angle. (See the documentation for model `scatteringPhaseFunctions` for more details.) The Legendre polynomials may be computed using a recurrence relation:

$$\begin{aligned} P_0(x) &= 1 \\ P_1(x) &= x \\ P_{l+1}(x) &= ((2l+1)xP_l - lP_{l-1})/(l+1) \end{aligned} \tag{2}$$

Using the module

Depends on modules: none.

Defines types: none.

Procedures: `computeLobattoTerms, computeGaussLegendreTerms, computeLegendrePolynomials, findIndex, gammln`

Procedures

Procedure computeLobattoTerms

Compute the weights and abscissae for a user-specified number of points on the interval [1, 1].

Definition

```
pure subroutine computeLobattoTerms(mus, weights)
  real, dimension(:), intent(out) :: mus, weights
```

Use

To compute the interval of a function `func(x)` over the interval [-1, 1] using `N`-point Lobatto integration

```
  real, dimension(Npoints) :: xs, weights
  …
  call computeLobattoTerms(xs, weights)
  sum = dot_product(weights(:), func(xs(:))
```

Procedure computeGaussLegendreTerms

Compute the weights and abscissae for a user-specified number of points on the interval (1, 1).

Definition

```
pure subroutine computeGaussLegendreTerms(mus, weights)
  real, dimension(:), intent(out) :: mus, weights
```

Use

To compute the interval of a function `func(x)` over the interval (-1, 1) using `N`-point Gauss-Legendre integration

```
  real, dimension(Npoints) :: xs, weights
  …
  call computeGaussLegendreTerms(xs, weights)
  sum = dot_product(weights(:), func(xs(:))
```

Procedure computeLegendrePolynomials

Compute the value of the Legendre polynomials at a set of user specified points in the interval [-1, 1]. This is an array-valued function. The first dimension is the order of the polynomial, starting with 0; the second dimension is the position in the interval.

Definition

```
pure function computeLegendrePolynomials(maxL, mus) result(legendreP)
  integer,            intent( in) :: maxL
  real, dimension(:), intent( in) :: mus
  real, dimension(0:maxL, size(mus)) :: legendreP
```

Use

Given a series of `L` coefficients `chi(:)` in a Legendre series expansion of a phase function, and a set of `N` points `x(:)`, the phase function at those points can be computed as

```
  allocate(legendreP(0:L, N))
  legendreP(:, :) = computeLegendrePolynomials(L, cos(x(:)))
  value(:) = matmul((/ 1., chi(:) /),                   &
                 spread( (/ (2*l + 1, l = 0, L) /), &
                         dim = 2, ncopies = nValues) * legendreP(:,:))
```

Procedure findIndex

Finds the index into a table such that `table(i) <= value < table(i+i)`. We make several assumptions: 1) that values in the table are always increasing, 2) that `value` will be spanned by the table entries, and 3) that `firstGuess` always makes sense (i.e. that is greater than 1 and less than the size of the table minus 1).

Definition

```
pure function findIndex(value, table, firstGuess)
  real,                intent( in) :: value
  real, dimension(:), intent( in) :: table
  integer, optional,  intent( in) :: firstGuess
  integer                          :: findIndex
```

Procedure gammln

Computes the natural log of the function $\Gamma(x)$.

Definition

```
elemental function gammln(x)
  real, intent(in) :: x
```

Module CharacterUtils

This module contains several utility functions for converting integers and real numbers to and from character strings. No error checking is done, so passing incorrect arguments to these functions is likely to cause a fatal Fortran runtime error.

Using the module

Depends on modules: none.

Defines types: `none`.

Procedures: `CharToInt, IntToChar, CharToReal`

Procedures

Procedure CharToInt

Reads a character string into an integer variable.

Definition

```
elemental function CharToInt(inputString)
  character(len = *), intent( in) :: inputString
  integer                         :: CharToInt
```

Procedure IntToChar

Writes an integer as a left-justified character string.

Definition

```
elemental function IntToChar(integerValue)
  integer,              intent( in) :: integerValue
  character(len = maxStringLength)  :: IntToChar
```

Procedure CharToReal

Reads a character string into a real variable.

Definition

```
elemental function CharToReal(inputString)
  character(len = *), intent( in) :: inputString
  real                            :: CharToReal
```